Decentralized Information Flow Control on a Bare-Metal JVM

Karthikeyan Manivannan Christian Wimmer Michael Franz
Department of Computer Science
University of California, Irvine
{kmanivan, cwimmer, franz}@uci.edu

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Languages

Keywords

Information flow control, Java virtual machine, Hypervisors, bare-metal jvm

1. INTRODUCTION

A large array of privacy sensitive applications like banking servers, medical records processors, and legal software are Java applications. Preserving user privacy is a necessary feature in such applications. For example, in a medical records system, only the authorized doctors and medical staff should be allowed access to patient information. Decentralized Information Flow Control (DIFC) [10] provides an effective means for preserving user privacy. In a traditional setup where the Java Virtual Machine (JVM) runs on top of an Operating System (OS), sensitive information flows both through the JVM and the OS, and effective enforcement of information flow policies requires tracking data across both these entities [12]. Implementing information flow control in such systems requires modification, and subsequent auditing, of both the JVM and OS source code.

In this paper, we provide a brief description of a design for implementing DIFC for Java, based on Guest VM [6] - a metacircular "Bare-metal" JVM (BMJVM) from Sun Microsystems. A BMJVM is a JVM that can run directly, on platform-virtualization software called *Hypervisors*, without an underlying OS. BMJVMs are becoming popular for hosting Java applications because of their higher server utilization and management benefits. Since our DIFC system is based on a metacircular BMJVM, i.e., a BMJVM implemented in Java, all the code that has to be audited is in a type-safe language. This arguably is easier to audit than the code of a traditional JVM-on-OS based information flow system, where the OS is usually written in a type-unsafe language like C/C++.

The remainder of this paper is organized as follows. Section 2 provides an introduction to BMJVMs. Section 3 describes the design of Guest VM. Section 4 introduces the concepts of DIFC. Section 5 provides a brief description of DIFC on GuestVM. Section 6 and Section 7 explain the related work and conclusion respectively.

CSIIRW '10, April 21–23, 2010, Oak Ridge, Tennessee, USA.

2. BARE-METAL JVM

Platform Virtualization using hypervisors has become a ubiquitous method for increasing server utilization and reducing costs. Hypervisors also provide benefits like easy administration, fault isolation and migration. Hypervisors typically run directly on the hardware and host several OSs, called *domains*, that share the hardware resources. A traditional OS, like Linux, has to support different kinds of applications which makes it difficult for it to be specialized for a specific program like the JVM. For instance, a JVM has its own memory protection and thread handling mechanism, which makes similar mechanisms in the OS redundant. In a typical setup, JVMs running on a virtualized machine would run inside OS instances which in turn run on the Hypervisor. In such a setup, performance could suffer because the OS has very little knowledge about the semantics of the JVM. For instance, the OS could preempt the Garbage Collector (GC) thread of a stop-the-world GC where application threads cannot progress until the GC thread finishes. Both the OS and Hypervisor implement page swapping, but neither of them have information about the memory usage pattern of the Java application.

BMJVMs solve these problems by pushing typical OS components like thread scheduling, memory management, networking stack and filesystem into the BMJVM and thus eliminating the need for an OS. Hypervisors provide a good platform for running BMJVMs - they abstract away the hardware and they provide lower-level thread scheduling and memory management APIs than an OS. BMJVMs also avoid the cost of context switching to and from the OS. Figure 1 depicts the architecture of a virtualized machine running a BMJVM and a traditional JVM. A BMJVM can also be designed without networking and filesystem services, like in the case of IBM J9 on Libra [1], where the BMJVM borrows these services from an OS domain running on the same hy-

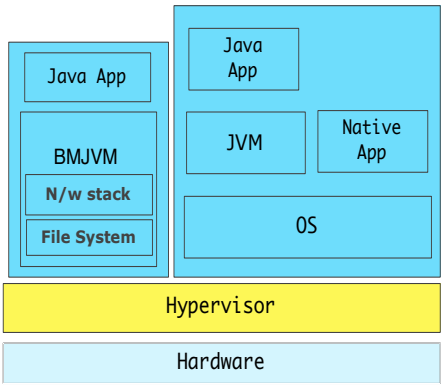

Figure 1: A virtualized machine running a BMJVM and a traditional JVM

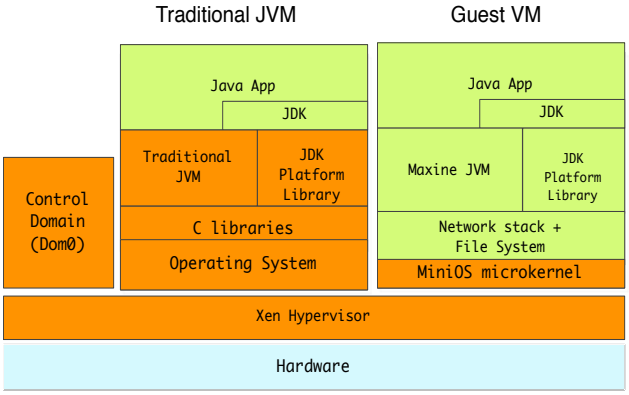

Figure 2: A virtualized machine running a control domain (Dom0), a traditional JVM on top an OS, and GuestVM. Components shaded in blue are written in Java and components shaded in orange are written in C/C++.

pervisor. Liquid VM [13], and Azul JVM [5] are examples of BMJVMs that have become a popular choice for hosting enterprise Java applications.

3. GUEST VM

Guest VM is a BJVM based on the Maxine JVM [8] and it runs on the Xen Hypervisor [2]. Xen can virtualize hardware to run multiple guest OSs. One of these OS instances, called *domain0* (Dom0) in Xen terminology, boots along with Xen and it has privileges to control other guest OS instances which are called *domainU* (DomU). Guest VM runs as a DomU guest and a traditional OS like Linux serves as the control domain (Dom0). Guest VM is metacircular and has an all Java software stack, i.e., components like the file system and the networking stack are written in Java. This provides an opportunity for the Maxine JVM's dynamic compiler to optimize the complete software stack from the application down to hypervisor API [6]. Maxine has a method substitution annotation that allows a native method call to be substituted by a call to a Java method. Since Guest VM implements the entire software stack in Java, it uses this substitution annotation to replace all the JDK platform related native methods with equivalent Java methods. Though Guest VM is mostly written in Java, its binary image is statically linked to an enhanced version of the C-based MiniOS microkernel. MiniOS handles startsup, scheduling and provides a block device driver to Guest VM. Guest VM has to use Dom0 disk drivers to write files to the disk. There is a connection from Guest VM to Dom0, through Xen, which is modeled like a default block device. Similarly, Guest VM uses the network interface drivers on Dom0 to communicate with the network interface. Figure 2 shows the detailed architecture of virtualized machine running a Guest VM instance, and an OS instance running a traditional JVM.

4. DECENTRALIZED INFORMATION FLOW CONTROL

The following section gives an introduction to DIFC as modeled by Meyers [10]. A DIFC system allows users to attach secrecy and integrity labels to data and track their flow through a system without the need for a centralized entity to enforce the secrecy and integrity policies. Security policies are associated with system entities like users, threads and processes. These entities are called *principals*. A secrecy policy restricts the use of data to only the principals that have sufficient authority. An integrity policy ensures that a principal reads only data slots that have sufficient credibility and that no writes occur to data slots with higher credibility than that of the principal. A *label* attached to a data has a set of policies which govern the use of that data.

A DIFC system ensures that information can flow between principals only as long as secrecy and integrity constraints associated with the corresponding label are satisfied. Principals also have *capabilities* which allow them to classify or declassify data. Each label L has a set of *owners*, owners are principals whose data was used in the creation of the data covered by L. Owners of a label can specify the principals that are authorized to read the data associated with the label. Principals with sufficient capabilities can modify flow policy for a label. For example, an owner can declassify data for principal P, by adding P to the set of readers. A principal P, can also grant another thread the capability to declassify data that P owns. Data owned by multiple principals can be declassified only if all the owners agree to declassify it. A principal can act on behalf of another principal if it has the appropriate privileges to do so. A principal does not have to go through a trusted central principal to declassify data or to grant another principal privileges that it owns.

An advantage of DIFC is that other principals do not have to trust a principal's declassification decisions, as a principal cannot dilute the security policies of principals that it does not act for [10]. DIFC can be implemented either at the language-level, at the OS-level or as a hybrid of both these techniques. Purely language based techniques are not good at tracking flow of information through OS resources like files and sockets. OS based techniques are not good at tracking information flow through program data structures. Hybrid techniques overcome these limitations by combining these two approaches. Section Section 6 talks about these techniques in detail.

Example. The following example shows how a hybrid DIFC works. Consider an e-commerce website where a user, A, can use a plugin, C, to find out the percentage of other users who purchased books that A purchased. A can give C access to read her entire purchase history, but ability to only declassify book-purchase history. All other participating users provide similar privileges to C. The similarity result that C computes will have security labels of book-purchase data from all the participants as it is derived from book-purchase history of all the users. This would make it impossible for C to reveal the result to A, as other users have not granted A the ability to read their book-purchase data. But since all participants have given C the ability to declassify their corresponding book-purchase data, C can combine these privileges to declassify the result and publish it to A. C cannot reveal a user's non-book purchase data to other users as it does not have the privileges to declassify non-book data. If C tries to create a new file with a user's complete purchase history, the file would automatically have all the labels associated with the user's purchase data, which would make it impossible for someone without these privileges to read the file. This example shows how principals do not have to go through a central entity to grant privileges to other principals and how principals cannot reveal user information without appropriate privileges.

5. DIFC ON GUEST VM

The following section describes the design for a Guest VM based hybrid DIFC system for Java. The system tracks information flow through program data structures, and OS resources like the file system and the networking stack. The system uses Guest VM threads as principals, and the label/capability system like in [11] and Laminar [12]. The user can specify security policies by defining lexically scoped secure regions with security labels and capabilities. Guest VM threads get initialized with certain labels and capabilities and these change dynamically based on the security regions that the thread executes. A *DIFC Monitor* inside the Guest VM ensures that a thread entering a security region acquires the capabilities provided by the region and only executes within the secrecy and integrity restrictions of the region.

Meyers introduces the concept of *runtime principals* [10], where security regions can use principals as types to model systems that are heterogenous with respect to principals. Since the file system and networking stack in Guest VM are written in Java, information flow policies for these entities can be specified using runtime principals. Figure 4 shows how file system code can use runtime principals to enforce information flow. The Reference Monitor can track OS resources just like it tracks program data structures. This eliminates the need for having a separate logic for DIFC enforcement on filesystem and network resources. DIFC systems like Laminar cannot make use of this technique as the filesystem and networking logic is part of the OS.

Unlike Laminar, where type-unsafe C/C++ code has to be modified to implement DIFC in the OS, the ability to specify security policies for filesystems and network stack in Java, removes the need to modify any type-unsafe C/C++ code. Since Guest VM uses C-based components like MiniOS, Xen and the Dom0 drivers to access the physical disk and network, one could argue that the type-unsafe code in these entities needs to be audited and trusted. These entities

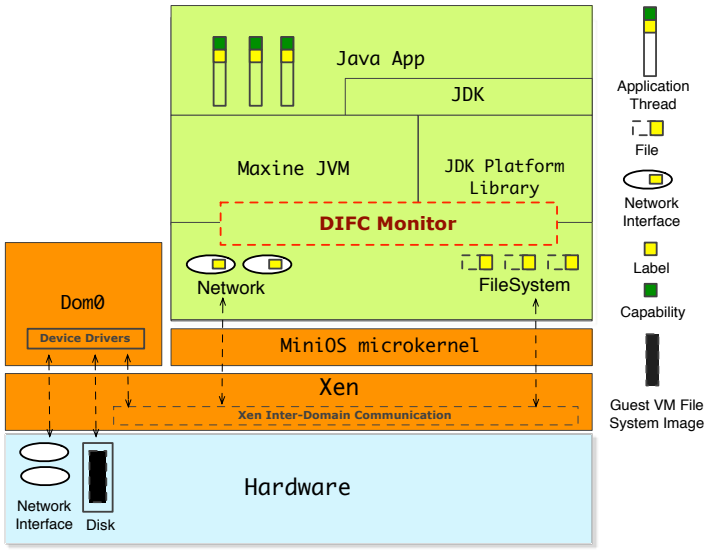

Figure 3: Architecture for a Guest VM based DIFC. The DIFC Monitor inside Guest VM handles DIFC for both program data structures and OS resources. Dom0 drivers are used to communicate with the disk and network interface.

can be removed from the trusted code base by encrypting file and network data within Guest VM before sending it out. If the Java application's disk usage is not high, the file system can be implemented as a RAM file system which gets encrypted and stored onto disk, either periodically, or before shutdown. Figure 3 shows the architecture for the Guest VM based DIFC system. Laminar, the only known functional hybrid DIFC, does not talk about what trust assumptions it makes about network and filesystem drivers. The absence of an underlying OS protects GuestVM from OS based intrusion attacks. For example, Laminar uses the Linux Security Module to enforce information flow policies at the OS level, this could expose the system to rootkit based attacks [4].

```
public int open (String name, int flags) {
    final principal userThread = Runtime.getUserThread();
    if (!DIFC.isUserAllowedOpen(userThread, name, flags)) {
        return Error.USER_NOT_ALLOWED;
   }
    ............
    ............
}
```

Figure 4: This figure shows an example of how information flow policies for files can be enforced using runtime principals inside filesystem code.

6. RELATED WORK

The idea of decentralizing information flow control was introduced by Meyers [10]. DIFC systems can be classified into three types based on the abstraction level used to implement DIFC.

OS based DIFC. Asbestos [3] implements a new OS that uses labeling and isolation mechanisms to provide information flow guarantees. HiStar [14] is a new OS that tries to minimize the the amount of trusted code while providing strict information flow control using labels and a page table based protection mechanism. Flume [7] uses a user-level reference monitor to track information flow at the granularity of processes. OS based techniques work well at the granularity of OS entities, but they either cannot, or are inefficient at tracking information flow through program data structures.

Programming Language based DIFC. Programming Language based DIFC systems like Jif [11] and Jflow [9] extend the type system of a language to allow static and runtime enforcement of information flow policies. These systems are good at detecting information flow violations at the level of program data structures, but cannot detect leaks in OS entities like files and network interfaces.

Hybrid DIFC. Laminar [12] allows the user to specify security policies for Java programs which it efficiently tracks through the JVM and the OS. Laminar augments the Jikes RVM to track information flow at the data structure level and it uses Linux Security Modules (LSM) to implement OS-level enforcement. The use of LSM could expose the system to sophisticated rootkit based attacks. The trusted code base also includes LSM code written in type-unsafe C/C++. In contrast, all the trusted code in a Guest VM based DIFC is in Java, which is arguably easier to audit than C/C++ code.

7. CONCLUSION

Hybrid DIFC systems allow effective implementation of DIFC across all layers of the software stack. Implementing DIFC on a BMJVM has the benefits of not having to modify and audit OS code. Implementing DIFC on a metacircular BMJVM allows us to write all the components in Java. Having filesystem and networking code implemented in Java allows uniform DIFC monitoring for both program data structures and OS resources. The trusted code base is completely within the DIFC enabled Guest VM, we argue that this facilitates easier audit and maintenance in comparison to a traditional hybrid DIFC system.

8. ACKNOWLEDGEMENTS

Parts of this effort have been sponsored by the California MICRO program and Industrial sponsor Sun Microsystems under Project No. 07-127, as well as by the National Science Foundation (NSF) under grants CNS-0615443 and CNS-0627747. Further support has come from unrestricted gifts from Sun Microsystems, Google and Mozilla, for which the authors are immensely grateful. The authors would like to thank the Guest VM and Maxine teams at Sun Labs for their guidance and support.

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and should not be interpreted as necessarily representing the official views, policies, or endorsements, either expressed or implied, of the NSF, any other agency of the U.S. Government, or any of the companies mentioned above.

9. REFERENCES

[1] Ammons, G., Appavoo, J., Butrico, M., Da Silva, D., Grove, D., Kawachiya, K., Krieger, O., Rosenburg, B., Van Hensbergen, E., Wisniewski, R.: Libra: a library operating system for a jvm in a virtualized execution environment. In: Proceedings of the 3rd international conference on Virtual execution environments, ACM (2007) 54

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM (2003) 177

[3] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazieres, D., Kaashoek, F., Morris, R.: Labels and event processes in the Asbestos operating system. ACM SIGOPS Operating Systems Review 39(5) (2005) 30

[4] grsecurity: grsecurity - LSM comments. http://www.grsecurity.net/lsm.php (2010) [Online; accessed 14-March-2010]

[5] Inc., A.S.: Azul compute appliances. http://www.azulsystems.com/products/compute_appliance.htm (2010) [Online; accessed 14-March-2010]

[6] Jordan, M.: Project Guest VM. http://research.sun.com/projects/guestvm/ (2010) [Online; accessed 14-March-2010]

[7] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M., Kohler, E., Morris, R.: Information flow control for standard OS abstractions. ACM SIGOPS Operating Systems Review 41(6) (2007) 334

[8] Mathiske, B.: The maxine virtual machine and inspector. (2008)

[9] Myers, A.: JFlow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM New York, NY, USA (1999) 228–241

[10] Myers, A., Liskov, B.: A decentralized model for information flow control. In: Proceedings of the sixteenth ACM symposium on Operating systems principles, ACM (1997) 142

[11] Myers, A., Zheng, L., Zdancewic, S., Chong, S., Nystrom, N.: Jif: Java information flow. Software release. Located at http://www. cs. cornell. edu/jif 2005 (2001)

[12] Roy, I., Porter, D., Bond, M., McKinley, K., Witchel, E.: Laminar: practical fine-grained decentralized information flow control. ACM SIGPLAN Notices 44(6) (2009) 63–74

[13] Systems, B.: Understanding LiquidVM. http://download.oracle.com/docs/cd/E13223_01/wls-ve/docs92-v11/config/lvmintro.html (2010) [Online; accessed 14-March-2010]

[14] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., Mazieres, D.: Making information flow explicit in HiStar. In: Proc. of the 7th OSDI. 263–278